下载APP

13｜哈夫曼树：HTTP2.0是如何更快传输协议头的？

2022-01-08 黄清昊

《算法实战高手课》 课程介绍 >

讲述：黄清昊
时长 16:22 大小 15.01M

你好，我是微扰君。

HTTP 是当今最广为使用的互联网传输协议，我们都听说过 HTTP/1.0、HTTP/2.0、SPDY、HTTP/3.0 等概念，但是对这几者之间的区别能如数家珍的同学却不多，比如 HTTP/2.0 在编码方面做了什么样的改进，比 HTTP/1.1 的传输更快呢？

领资料

我们今天就来学习一下 HTTP/2.0 为了提高传输效率而引入的用于头部压缩的杀招：**HPACK**。

HPACK 应用了静态表、动态表和哈夫曼编码三种技术，把冗余的 HTTP 头大大压缩，常常可以达到 50% 以上的压缩率。其中的哈夫曼编码，底层主要就依赖了我们今天会重点学习的哈夫曼树，这也是广泛运用在各大压缩场景里的算法。

在展开讲解 HTTP/2.0 中的 HPACK 到底是怎么工作的，我们首先要来思考一下为什么要压缩 HTTP 的头，或者说，压缩到底又是什么呢？

压缩技术

我们都知道压缩技术诞生已久，在各种文件尤其是多媒体文件里，应用非常广泛，能帮助节约信息的存储空间和网络传输时间。

之所以能压缩，主要原因就是我们存储的信息往往是有模式和冗余的。以文本为例，大量单词的重复或者大量的空格，都是我们可以压缩的空间。原文件大小与压缩后文件大小的比值，我们就叫做压缩比，是衡量压缩算法有效性非常直观的指标。

压缩技术也分为有损压缩和无损压缩两种。

有损压缩，我们允许数据一定程度上的丢失，它在多媒体文件里更加流行，比如 JPEG、MP3 就是非常典型的两种数据有损压缩的方式。

压缩多媒体数据时，我们允许一定程度的丢失。主要是因为对图像或者音频文件来说，数据一定程度上的丢失并不一定会很影响用户的主观感受。比如压缩图片时，有一种方式会把颜色的种类减少，让图片每个像素的编码位数降低，从而就实现了图片的压缩，但是从人的视觉上说影响可能并不是特别大。所以有损压缩的衡量指标就不止压缩比，还需要考虑人的主观感受了。

但是互联网大部分应用中所用的通信协议，都不应该关心业务数据本身，要做的只是保证数据可以按照一定的方式准确、无误，并且尽量高效地从发送端传输到接收端，有损压缩显然是不可接受的。比如最常用的 HTTP，就不会关心具体传输的内容是什么，自然不可能对数据做有损压缩。

所以在 HPACK 里，我们采用的当然也是无损压缩策略。

现在搞清楚了压缩技术的背景，在那 HTTP 里引入压缩技术是否有意义呢？毕竟如果压缩比不是很高，引入这样的设计，只会导致相关协议的客户端和服务端实现的复杂性提高，得不偿失。

引入 HPACK 的价值

早期采用 HTTP 的互联网应用，只涉及数据的展示，所以我们最初设计 HTTP 的时候没有引入状态，但是后期随着 Web2.0 的繁荣发展，网站不再只是展示这么简单，会和用户产生更多的交互，让用户产生内容，于是也引入了“登录”等有状态的功能。

要基于 HTTP 实现相关应用，我们常用的做法就是把用于鉴权的令牌或者其他“状态”携带在 HTTP 头里，在客户端和服务端之间来回传递。

出于安全需要，这种鉴权的令牌往往非常冗长，http header 常常比 http body 还要大，这就带来了**很大的开销**。所以 HTTP/2.0 通过引入 HPACK 压缩协议头，就带来了很大的价值。

而且 HTTP/1.1 之前的 HTTP 协议传输的内容很简单，可以认为就是一串文本在互联网上直接传递，没有任何编码，这也给我们的压缩算法带来了**很大的压缩空间**。

HPACK 的压缩效果

既然 HTTP 中引入压缩技术很有意义，那我们就先来看看压缩之后的效果到底有多大吧。

h2load 是网上开源的一个对 HTTP/2.0 做 benchmark 的工具，我们可以在系统上安装它，来访问某些网站，直观地感受一下 HPACK 技术带来的 HTTP 头的大小变化：

```
→  Perturbation h2load -n10 https://leetcode-cn.com/u/wfnuser
starting benchmark...
spawning thread #0: 1 total client(s). 10 total requests
TLS Protocol: TLSv1.2
Cipher: ECDHE-RSA-AES128-GCM-SHA256
Server Temp Key: X25519 253 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 1.02s, 9.83 req/s, 3.68KB/s
requests: 10 total, 10 started, 10 done, 10 succeeded, 0 failed, 0 errored, 0 timeout
status codes: 0 2xx, 10 3xx, 0 4xx, 0 5xx
traffic: 3.75KB (3838) total, 1.73KB (1769) headers (space savings 33.25%), 1.71KB (1750) data
                     min         max         mean         sd        +/- sd
time for request:    48.98ms    259.82ms     86.39ms     66.71ms    90.00%
time for connect:   153.05ms    153.05ms    153.05ms         0us   100.00%
time to 1st byte:   412.92ms    412.92ms    412.92ms         0us   100.00%
req/s           :       9.83        9.83        9.83        0.00   100.00%
```

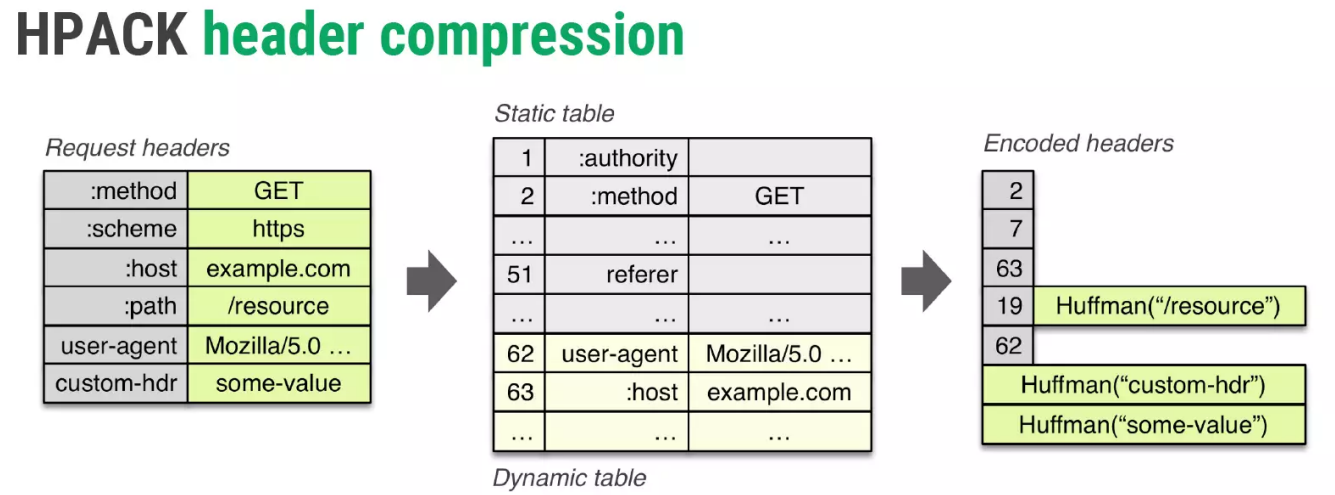
可以看到，采用了 HTTP/2.0 之后，直接压缩了 HTTP 头 33% 的空间，效果显著。

那 HPACK 具体是如何做的呢？我们现在就来一探究竟。

1.HPACK 中的静态表

首先我们来看一下 HPACK 的第一个手段：静态表，它其实就是对 HTTP 头报文里最常见的文本进行了一种编码。静态表也是非常常用的压缩手段。

HTTP/2.0 一共对 61 个常用的头，以及头和值的组合做了编码（图片来源）。



比如 HTTP 的几种请求方法，GET、POST 等，都编码到了一张范围为 1-61 的索引表里，这样原来的":method GET"等字符串需要的空间就小多了。

2.HPACK 中的动态表

但是只是如此的话，我们能压缩的报文就非常有限了，怎么办呢？

我们应该还**允许客户端和服务端，通过通信的方式，维护一张动态的“字典”**，这样用索引号就可以代表一串很长的文本，减少在这次 HTTP/2 连接里反复出现的一些自定义字段的载荷。

比如，这些字段就是很好的例子：

```
1 :authority wfnuser
2 :cookies xxxxxx
```

尤其是常常用来保留用户身份凭证的cookies，因为安全性和加密算法的需要，它们往往设计的比较长，很多时候甚至导致 header 的长度比 body 还要长。

在 http/1.1 之前的协议里，每次通信都需要传递冗长且重复的信息，显然会带来巨大的开销。动态表就很好地解决了这个问题。

3.HPACK 中的哈夫曼编码

其实只用动态表和静态表已经可以做到很好的压缩 header 了，但是受限于静态表和动态表的大小，我们并不能用它们压缩任意字符。

哈夫曼编码，就是对静态表和动态表能力的一种补充，HPACK 在引入了哈夫曼编码之后，可以达到对 HTTP 报文高达 30%-80% 的压缩率。

那我们首先来了解一下哈夫曼编码是什么。

哈夫曼编码

其实，哈夫曼思想非常简单，就是让**出现概率更高的字符用更短的编码表示，出现概率低一些的字符则用更长的编码表示。**

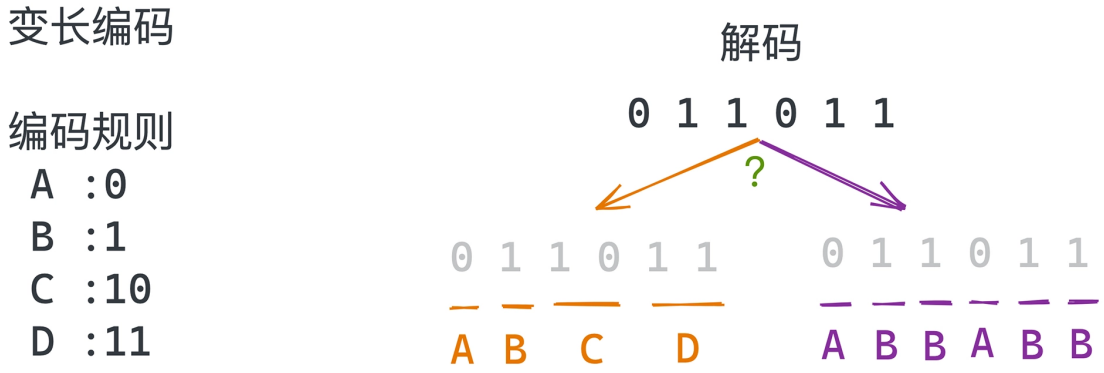
这句话乍一听可能不太好理解，什么叫更短的编码呢，或者说什么是编码呢？

我们日常在用的 ASCII 编码就是对字符串的一种编码，每个字符都被编码到 0-127 的范围里，这也是在绝大部分编程语言里，一个 char 类型的字符只占用一个字节的原因；当然，一个字节实际可以表示 0-255 种可能，ASCII 编码规范本身没有定义 128-255 的范围，所以各大厂商都可以有自己的扩展定义去表示更多的字符。

所以 ASCII 作为一种典型的每个字符都等长编码的编码方法，有没有办法被压缩呢？是可以的。

比如，在自然语言场景里，我们知道字母 e 可能是最常出现的字符，如果用更短的编码去表示 e，而用其他更长的编码表示其他字符，就可以达到压缩文本编码长度的效果。

不过这里还有个问题，用等长编码比如 ASCII 编码，我们解码的时候直接 8 位、8 位的读，就很容易解出编码前的字符串，但**如果用变长编码，我们就需要处理解码歧义的问题。**

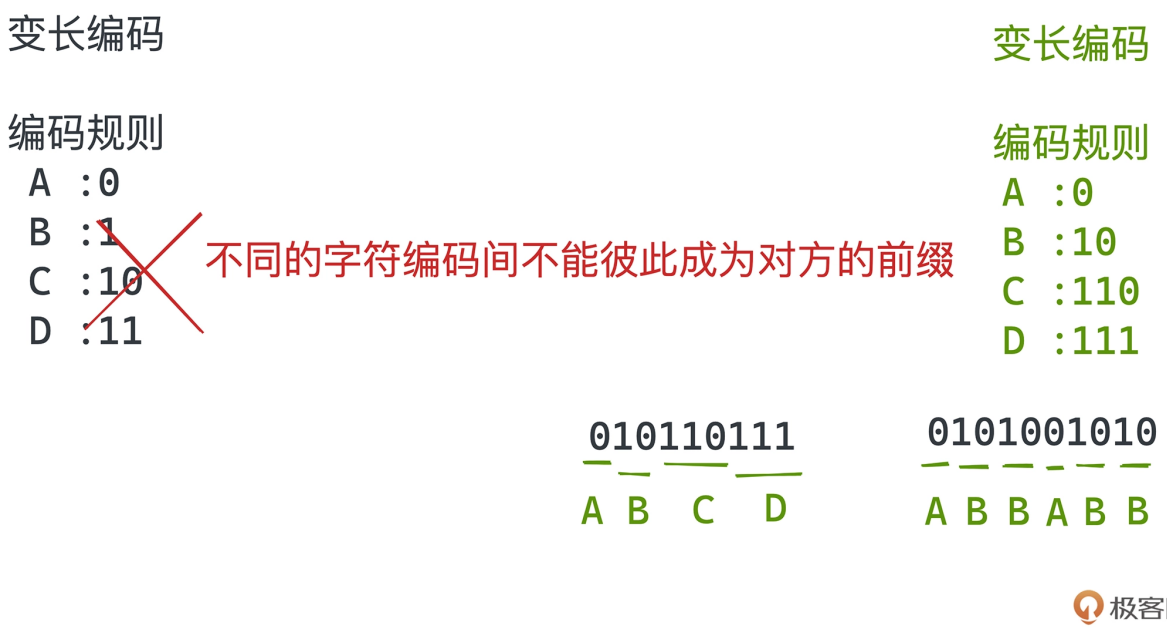


比如在字符串“ABCD”里，我们假设把 A 用 0 编码、B 用 1 编码、C 用 10 编码、D 用 11 编码。“ABCD”可以编码成二进制编码为“011011”，没问题，但如果中间不加任何分隔符，你并不能知道这个编码结果是由“ABCD”产生的还是由“ABBABB”产生的。

如果加了分隔符，分隔符本身也会引入额外的编码成本，甚至可能导致一个负向的优化。

为了解决这个问题，学生时代的哈夫曼（huffman）在 1952 年提出了最优前缀码的算法，也就是广泛应用在压缩领域的哈夫曼编码，它除了用更短的编码表示出现概率更高的字母，还引入了一个约束：**不同的字符编码间不能彼此成为对方的前缀。**

这条约束在解码的时候完美地避免了歧义的问题。比如刚刚如果把 A 编码成 0、B 编码成 10、C 编码成 110、D 编码成 111，这就是一种符合约束的前缀码编码方式。ABCD 就会编码成 010110111，一定只有一种解码方式。



那在不能成为对方前缀的约束下，具体如何根据出现频率选择合适的编码方式呢？

哈夫曼采用了**贪心的算法思想：用一棵二叉树来标记每个字符的编码方式，左分支代表 0、右分支代表 1，所有需要编码的字符都对应二叉树的叶子节点，根结点到该叶子结点的路径就代表着该字符的编码方式。**由于各节点是独立的不可能重复，每个字符又都唯一对应着一个叶子节点，所以它们一定不会互相成为对方的前缀。

下面我们要做的就是找到这样一个可以达到最大压缩效率的二叉树。

贪心的哈夫曼树

让我们举一个例子来理解哈夫曼树的编码方式。假设要对 a b c d e f 进行编码，它们在需要编码的文本中出现的频率分别是 5 9 12 13 16 45。

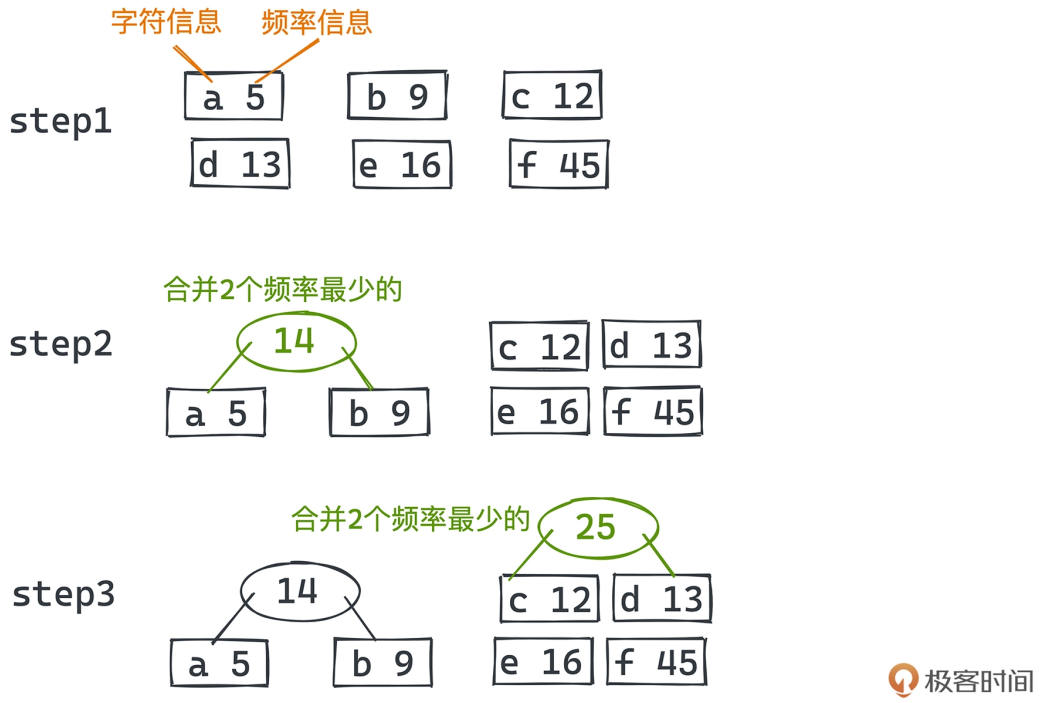
复制代码

```
a    5
b    9
c    12
d    13
e    16
f    45
```

那么如何编码可以让整个文本编码出来的二进制所占空间最少呢？

最开始，我们先把每个字符都看成一个独立的二叉树节点，节点中同时包含了字符信息和频率信息。

然后，从中选两个出现频率最少的节点，a 和 b，我们把这两个节点合并成一棵子树，也就是用一个父节点的左右节点指针分别链向 a 和 b 两个节点，把两者的频率之和作为父节点的频率。



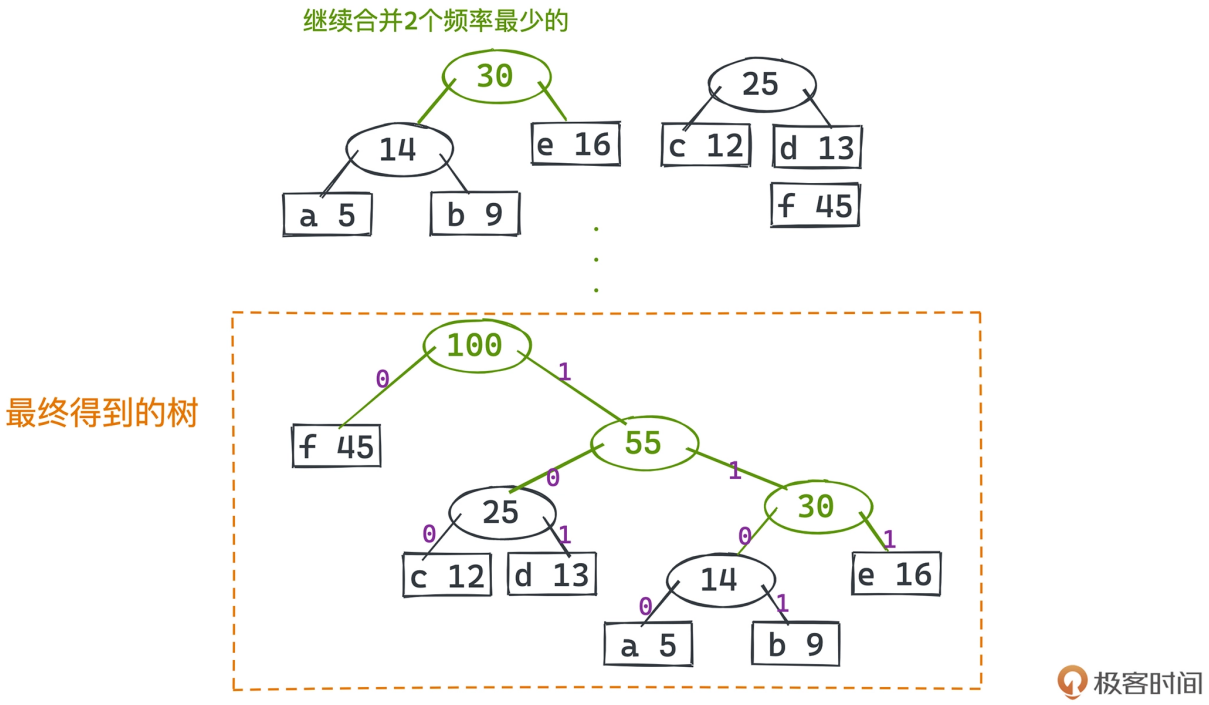
之后，我们把这个频率为 14 的树放回所有的节点里（14 12 13 16 45），再从中继续选择最小的两个节点 c 和 d 合并成一棵新的树。

更新之后的所有节点就是下面这些，其中 a、b、c、d 节点都被替换成了新的合成节点：

复制代码

```
1    (a, b)    14
2    (c, d)    25
3    e         16
4    f         45
```

不断进行这样的操作，最终就可以得到这样一棵树：



假设树的左链代表 0，右链代表 1，a b c d e f 对应的编码为：

复制代码

```
1    f        0
2    c        100
3    d        101
4    a        1100
5    b        1101
6    e        111
```

看完这个过程，相信你对为什么这样编码也有一些想法了，它非常直观，我们来一起研究一下。

> 首先为了不用额外的分隔符，一种让解码不产生歧义的办法就是引入前缀码规则，对应到树上就是每个字符都编码成根节点到某个叶子节点的路径。
>
> 然后为了“出现频率最高的字符用最短的方式编码”的策略，显然，我们需要让出现频率最高的树出现在最短的路径里，出现频率最低的树则放到更长的路径里。

因为每次将两颗树合并到一起时，都会导致这两颗树里所有叶子节点的高度加 1，也就是其中所有的字符编码长度都会 +1，所以，为了达到最优编码的目标，我们会从出现频率最低的节点开始合并。最后我们合并完新的树，也要把新树的频率变成这两颗树的频数之和，它代表了这颗树下所有字符出现的频率。

这样每次找出最低的两个树合并，就必然能得到一个整体最优的编码方式，也就是哈夫曼编码的思路了。

这背后的思想其实就是贪心算法，也就是在每一次决策时都采取在当前状态下最优的选择，从而得到整体最优解的算法。当然，也不是所有的场景都能使用贪心算法的，比如经典的背包问题，采用贪心算法虽然能得到局部最优解，但就不能得到全局最优解。而哈夫曼树则是一个贪心算法发挥作用的很好的例子。

哈夫曼树实现

现在有了思路，相关的实现其实就非常简单了。

先说编码的部分，实质就是要建立这样一棵基于贪心算法的哈夫曼树。二叉树的相关概念相信你已经非常熟悉了，所以这里的核心就是如何做到每次都可以快速选出频率最低的两棵树。

怎么做呢？相信你已经想到了，我们之前学的堆就是用来维护动态序列中最小值的利器。

假设一共要对 N 个节点进行编码，堆中最多有 N 个节点，每次合并涉及两次取出元素和一次放回元素，时间复杂度都是 O(logN)，整体时间复杂度为 O(N*logN)。翻译成 C++ 代码，我写了比较详细的注释供你参考：

复制代码

```
1     void buildHuffmanTree(string text)
2   {
3     // 利用hashmap对字符串进行频率计数
4     unordered_map<char, int> freq;
5     for (char ch: text) {
6       freq[ch]++;
7     }
8
9     // 用堆去动态维护所有树中最小的两颗
10    priority_queue<Node*, vector<Node*>, comp> pq;
```

```
11
12      // 将所有的字符都初始化成为哈夫曼树的一个叶子节点
13          // 并推入优先队列
14      for (auto pair: freq) {
15        pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
16      }
17
18      // 每次取出最小的两个合并 直至优先队列只剩一个节点
19      while (pq.size() != 1)
20      {
21        // 最小的两个节点出队
22        Node *left = pq.top(); pq.pop();
23        Node *right = pq.top();  pq.pop();
24
25        // 建立一个内部节点，以这两个最小的树为左右节点
26          int sum = left->freq + right->freq;
27        pq.push(getNode('\0', sum, left, right));
28      }
29
30      // 优先队列中最后一个元素为整棵树的根节点
31      Node* root = pq.top();
32    }
```

好，到这里我们就构建好了一棵哈夫曼树了。基于它，我们可以很方便地通过对哈夫曼树的遍历做到对字符串的编码和解码，进而实现压缩的效果。具体的实现都贴在GitHub仓库中了。

学习完哈夫曼编码，HPACK 中用于压缩的最后一个杀招你也就学会了。

HPACK 采用的是静态 huffman 编码，HTTP/2.0 协议制定者利用一个很大的 HTTP Header 的 sample，统计了所有字符出现的频率，并基于此构建了一个 huffman 编码表，需要内置在服务端和客户端里，最多能带给我们大约 37.5% 的压缩率。

总结

今天关于 HPACK 的讲解到这里就结束了，我们做个简单的小结。

HPACK 是 HTTP/2.0 为了降低 HTTP payload 大小从而提高传输效率的杀招，应用了静态表、动态表和哈夫曼编码三种技术，把冗余的 HTTP 头信息大大压缩，常常可以达到 50% 以上的压缩率。

前两招静态表和动态表的思想其实非常常见。

比如在设计消息系统时，微服务架构下经常涉及消息在不同系统间传递的需求，如果只是为了定位消息而不用真的读取消息体，我们完全可以把消息编码成“消息 ID + 消息体”的格式，存储在数据库或者其他缓存系统中，这样，在系统间传递的时候只需要传递 ID 即可，等真的需要取出消息体的时候，再到数据库等系统里读取具体内容。这可以大大减少系统通信的开销，背后其实就是类似动态表的思想，你可以举一反三。

第三招哈夫曼编码，引入不同的字符编码间不能彼此成为对方前缀的约束下，使用哈夫曼树来编码。哈夫曼树基于贪心的思想，以及用树对编码进行抽象的想法，也非常精巧，也值得你好好学习一下。

课后作业

最后再给你留个课后小问题，哈夫曼树在 HTTP/2.0 中一定可以获得更优的编码方式吗？为什么？欢迎你在留言区与我一起讨论。

参考资料

HPACK 主要由静态表、动态表和哈夫曼编码三个部分组成，具体可以参见 rfc7541。相信如果是对计算机网络熟悉的同学一定会了解 RFC 文档，主流的网络协议比如 WebSocket、MQTT、TCP 等等都有对应的 RFC 文档，它也是你学习相关协议最权威的参考资料之一。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 拓扑排序：Webpack是如何确定构建顺序的？

精选留言 (1)

写留言

Paul Shan

2022-01-08

如果所有字母出现的概率都相等，哈夫曼编码压缩就失败了，好像也没有什么编码能够处理这种情况。

作者回复: 是的 已经是熵最大的情况了